Java

Strings
String

• instances of `java.lang.String`
• compiler works with them *almost* with primitive types
  – String constants = instances of the String class

• immutable!!!
  – for changes – classes `StringBuffer`,
    `StringBuilder`

• operator +
  – String concatenation
  – if there is at least a single String in an expression -> all
    is converted to Strings and concatenated
    • method `toString()`
      – defined in the class Object
      – commonly overridden
  – creates a new String
java.lang.String

• constructors

String();
String(char[] value);
String(byte[] bytes);
String(byte[] bytes, String charsetName);
String(String value);
String(StringBuffer value);
String(StringBuilder value);
java.lang.String

• methods
  - int length();
  - char charAt(int index);
    • IndexOutOfBoundsException
  - boolean equals(Object o);
    • compares Strings
    • == compares references

String a = new String("hello");
String b = new String("hello");
System.out.println(a==b); // false
System.out.println(a.equals(b)); // true
java.lang.String

• methods
  - int compareTo(String s);
    • lexicographical comparison
  - int compareToIgnoreCase(String s);
  - int indexOf(char c);
  - int indexOf(String s);
    • return -1, if there is no such char or substring
  - String substring(int beginIndex);
  - String substring(int beginIndex, int endIndex);
  - String replaceFirst(String regexp, String repl);
  - String replaceAll(String regexp, String repl);
Strings

• methods (cnt.)
  - String join(CharSequence delimiter, CharSequence... elements);

  • since Java 8

• methods can be called on String constants also

```java
String s;
...
if ("ahoj".equals(s)) {
  ...
```
Java

Wrapper types
Wrappers

- immutable
- Integer
  - constructors – deprecated since Java 9
    - `Integer(int value)`
    - `Integer(String s)`
  - methods
    - `int intValue()`
    - `static Integer valueOf(int i)`
    - `static int parseInt(String s)`
    - ...

- other wrapper types similarly
More about methods
Local variables

• definition anywhere in body

• visible in a block
  – see the first lecture

• no initialization

• can be defined as final
  – constants
  – no other modifier can be used

• effectively final
  – defined without final but the value is never changed after it is initialized
Method overloading

• several methods with the same name but different parameters
  – different number and/or type

```java
public void draw(String s) {
    ...
}
public void draw(int i) {
    ...
}
public void draw(int i, double f) {
    ...
}
```

• cannot overload just by a different return type
Recursive calls

• recursion – a method calls itself

```java
public static long factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

• be aware about termination

• non terminated -> stack overrun
  – a size of the stack can be set
Java

Exceptions
Exceptions

- errors reporting and handling
  - an exception represents an error state of a program
- exception = an instance of \texttt{java.lang.Throwable}
- two subclasses – \texttt{java.lang.Error} and \texttt{java.lang.Exception}
  - specific exceptions – children of the above two classes
- \texttt{java.lang.Error}
  - "unrecoverable" errors
  - should not be caught
  - e.g. \texttt{OutOfMemoryError}
- \texttt{java.lang.Exception}
  - recoverable errors
  - should (has to) be caught
  - e.g. \texttt{ArrayIndexOutOfBoundsException}
Exception handling

• **statement** `try/catch/finally`

```java
try {
    // a block of code where an exception can happen and we want to handle it
} catch (Exception1 e) {
    // handling of exceptions with the Exception1 type and its subtypes
} catch (Exception2 e) {
    // handling of exceptions with the Exception2 type and its subtypes
} finally {
    // executes always
}
```
Exception handling

- if the exception is not caught in a block where it occurs, it propagates to the upper block
- if the exception is not caught in a method, it propagates to the calling method
- if the exception reaches `main()` and it not caught, it terminates the virtual machine
  - information about the exception is printed
try/catch/finally

- catch or finally can be omitted
  - but both cannot be omitted
Extended try (since Java 7)

• interface AutoClosable and extended try
  – example:
    ```java
    class Foo implements AutoClosable {
        ...
        public void close() { ... }
    }
    try ( Foo f1 = new Foo(); Foo f2 = new Foo() ) {
        ...
    } catch (...) {
        ...
    } finally {
        ...
    }
    – at the end of try (normally or by an exception),
      close() is always called on all the objects in the try declaration
    • called in the reverse order than declared
Extended try

- both catch and finally can be omitted together

```java
try (Resource r = new Resource()) {
    ...
}
```

- since Java 9, (effectively) final variables can be used in extended try

```java
final Resource resource1 = new Resource("res1");
Resource resource2 = new Resource("res2");

try (resource1; resource2) {
    ...
}
```
class Exception1 extends Exception {}
class Exception2 extends Exception {}

try {
    boolean test = true;
    if (test) {
        throw new Exception1();
    } else {
        throw new Exception2();
    }
} catch (Exception1 | Exception2 e) {
    ...
}
Exception declaration

• a method that can throw an exception must either
  – catch the exception, or
  – declare the exception via `throws`

```java
public void openFile() throws IOException {
  ...
}
```

• it is not necessary to declare following exceptions
  – children of `java.lang.Error`
  – children of `java.lang.RuntimeException`
    • it extends `java.lang.Exception`
    • `ex. NullPointerException`, `ArrayIndexOutOfBoundsException`
Throwing exceptions

- **statement** `throw`
  - throws (generates) an exception
  - "argument" – a reference to `Throwable`

```java
throw new MojeVyjimka();
```

- existing exceptions can be thrown but, commonly, own ones are used
- exceptions can be “re-thrown”

```java
try {
    ...
} catch (Exception e) {
    ...
    throw e;
}
```
Re-throwing (in Java 7)

class Exception1 extends Exception {}
class Exception2 extends Exception {}

public static void main(String[] args) throws Exception1, Exception2 {
    try {
        boolean test = true;
        if (test) {
            throw new Exception1();
        } else {
            throw new Exception2();
        }
    } catch (Exception e) {
        throw e;
    }
}
java.lang.Throwable

- has the field (private) typed String
  - contains a detailed description of the exception
  - method String getMessage()

- constructors
  - Throwable()
  - Throwable(String mesg)
  - Throwable(String mesg, Throwable cause)  // since 1.4
  - Throwable(Throwable cause)  // since 1.4

- methods
  - void printStackTrace()
public class MyException extends Exception {
    public MyException() {
        super();
    }
    public MyException(String s) {
        super(s);
    }
    public MyException(String s, Throwable t) {
        super(s, t);
    }
    public MyException(Throwable t) {
        super(t);
    }
}
Chains of exceptions

... try { .....
...
} catch (Exception1 e) {
...
throw new Exception2(e);
}
...

• throwing an exception as a reaction to another exception
  – it is common
    • reacting to a “system” exception by an “own” one
Suppressing exception

- in several cases an exception can suppress another one
  - it is not chaining of exceptions!
  - typically it can happen
    - if an exception occurs in the `finally` block
    - in the extended `try` block (Java 7)
- `Throwable[] getSuppressed()`
  - method in `Throwable`
  - returns an array of suppressed exceptions
Inner classes
Inner classes

• defined in the body of another class

```java
public class MyClass {
    class InnerClass {
        int i = 0;
        public int value() { return i; }
    }
    public void add() {
        InnerClass a = new InnerClass();
    }
}
```
Inner classes

• the inner class can return a reference to the outer class

```java
public class MyClass {
    class InnerClass {
        int i = 0;
        public int value() { return i; }
    }
    public InnerClass add() {
        return new InnerClass();
    }
    public static void main(String[] args) {
        MyClass p = new MyClass();
        MyClass.InnerClass a = p.add();
    }
}
```
Hiding inner class

- inner class can be `private` or `protected`
- access to it via an interface

```java
public interface MyIface {
    int value();
}

public class MyClass {
    private class InnerClass implements MyIface {
        private i = 0;
        public int value() {return i;}
    }
    public MyIface add() {return new InnerClass();}
}

... public static void main(String[] args) {
    MyClass p = new MyClass();
    MyIface a = p.add();
    // error - MyClass.InnerClass a = p.add();
}
Inner classes in methods

- an inner class can be defined in method or just a block of code
- visible just in the method (block)

```java
public class MyClass {
    public MyIface add() {
        class InnerClass implements MyIface {
            private i = 0;
            public int value() {return i;}
        }
        return new InnerClass();
    }
    public static void main(String[] args) {
        MyClass p = new MyClass();
        MyIface a = p.add();
        // error - MyClass.InnerClass a = p.add();
    }
}
```
Anonymous inner classes

```java
public class MyClass {
    public MyIface add() {
        return new MyIface() {
            private i = 0;
            public int value() {return i;}
        };
    }

    public static void main(String[] args) {
        MyClass p = new MyClass();
        MyIface a = p.add();
    }
}
```
Anonymous inner classes

```java
public class Wrap {
    private int v;
    public Wrap(int value) { v = value; }
    public int value() { return v; }
}

public class MyClass {
    public Wrap wrap(int v) {
        return new Wrap(v) {
            public int value() {
                return super.value() * 10;
            }
        };
    }

    public static void main(String[] args) {
        MyClass p = new MyClass();
        Wrap a = p.wrap(5);
    }
}
```
Anon. inner classes: initialization

- elements outside an anon. in. class necessary in the anon. in. class – final
- without final – compile-time error
- since Java 8 - “effectively” final is enough
  - i.e. declared without the final modifier, but there are no changes to the particular element

```java
public class MyClass {
    public MyIface add(final int val) {
        return new MyIface() {
            private int i = val;
            public int value() {return i;}
        };
    }
}
```

- till Java 7 final is necessary here
- since Java 8 final can be omitted
  - as there are no changes to val
Anon. inner classes: initialization

- anon. inner classes cannot have a constructor
  - because they are anonymous
- object initializer

```java
public class MyClass {
    public MyIface add(final int val) {
        return new MyIface() {
            private i;
            {
                if (val < 0)
                    i = 0;
                else
                    i = val;
            }
            public int value() {return i;}
        };
    }
```
Relation of inner and outer class

- the instance of an inner class can access all elements of the instance of the outer class

```java
interface Iterator {
    boolean hasNext();
    Object next();
}

public class Array {
    private Object[] o;
    private int next = 0;
    public Array(int size) {
        o = new Object[size];
    }
    public void add(Object x) {
        if (next < o.length) {
            o[next] = x;
            next++;
        }
    }
}  // cont....
```
Relation of inner and outer class

// cont....
private class AIterator implements Iterator {
    int i = 0;
    public boolean hasNext() {
        return i < o.length;
    }
    public Object next() {
        if (i < o.length)
            return o[i++];
        else
            throw new NoNextElement();
    }
}

public Iterator getIterator() {
    return new AIterator();
}

Relation of inner and outer class

- a reference to the instance of the outer class
  - OuterClassName.this
  - previous example – classes Array and AIterator
    - the reference to the instance of Array from
      Array.AIterator – Array.this
Relation of inner and outer class

• creation of the instance of an inner class outside of its outer class
  ```java
  public class MyClass {
    class InnerClass {
    }
    public static void main(String[] args) {
      MyClass p = new MyClass();
      MyClass.InnerClass i = p.new InnerClass();
    }
  }
  ```

• an instance of an inner class cannot be created without an instance of its outer class
  - instances of an inner class always have a (hidden) reference to an instance of its outer class
Inner classes in inner classes

- from an inner class, an outer class on any level of nesting can be accessed

```java
class A {
    private void f() {}
    class B {
        private void g() {}
        class C {
            void h() {
                g();
                f();
            }
        }
    }
}

public class X {
    public static void main(String[] args) {
        A a = new A();
        A.B b = a.new B();
        A.B.C c = b.new C();
        c.h();
    }
}
```
Inheriting from inner classes

• a reference to an instance of the outer class has to be explicitly passed

```java
class WithInner {
    class Inner {}
}
class InheritInner extends WithInner.Inner {
    InheritInner(WithInner wi) {
        wi.super();
    }
    // InheritInner() {} // compile-time error

    public static void main(String[] argv) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
}
```
Nested classes

- defined with the keyword `static`
- do not have a reference to an instance of its outer class
- can have static elements
  - inner classes cannot have static elements
- do not need an instance of the outer class
  - they do not have the reference to it
- in fact, they are regular classes just placed in the namespace of the outer class

```java
public class MyClass {
    public static class NestedClass {
    }
}

public static void main(String[] args) {
    MyClass.NestedClass nc = new MyClass.InnerClass();
}
```
Nested classes

- can be defined in an interface
  - inner classes cannot be

```java
interface MyInterface {
    static class Nested {
        int a, b;
        public Nested() {}
        void m();
    }
}
```
Inner classes and .class files

- inner (or nested) class – own .class file
  - OuterName$InnerName.class
    - MyClass$InnerClass.class

- anonymous inner classes
  - OuterName$SequentialNumber.class
  - MyClass$1.class

- a nested class can have the main method
  - launching: java OuterName$NestedName
Reasons for using inner classes

- hiding an implementation
- access to all elements of the outer class
- “callbacks”
- ...
Source files
Unicode

• programs ~ Unicode
  – comments, identifiers, char and string constants
  – the rest is in ASCII (<128)
    • or Unicode escape sequences < 128
• Unicode escape sequences
  – \uxxxxx
  – \u0041 .... A

• the expanded sequence is not used for following ones
  – \u005cu005a results in six chars
    • \ u 0 0 5 a
1. translation of unicode escape sequences (and all of the source code) into a sequence of unicode chars

2. the sequence from (1) is translated into a sequence of chars and line-terminating chars

3. the sequence from (2) is translated into a sequence of input tokens (without white-spaces and comments)

- line-terminating chars
  - CR LF
  - CR
  - LF
public class Test {
    public static void main(String[] argv) {
        int i = 1;
        i += 1; // is the same as \u000A i = i + 1;
        System.out.println(i);
    }
}

- Program prints out:
  a) 1
  b) 2
  c) 3
  d) cannot be compiled
  e) a runtime exception
Encoding

- argument of javac `-encoding`
  - encoding of source files
  - without it – default encoding
Literals

- integer literals
  - decimal ... 0 1 23 -3
  - hexadecimal ... 0xa 0xA 0x10
  - octal ... 03 010 0777
  - binary ... 0b101 0B1001
    - since Java 7
    - by default of the int type
      - long ... 1L 33l 077L 0x33L 0b10L
- floating-point literals
  - 0.0 2.34 1. .4 1e4 3.2e-4
  - by default double
    - float ... 2.34f 1.F .4f 1e4F 3.2e-4f
- boolean literals
  - true, false
• underscores in numerical literals
  – since Java 7
  – for better readability

1234_5678_9012_3456L
999_99_9999L
3.14_15F
0xFF_EC_DE_5E
0xCAFE_BABE
0x7fff_ffff_ffff_ffffL
0b0010_0101
0b11010010_01101001_10010100_10010010
Literals

• char literals
  - 'a'  '%'  '\\'  ''  '\u0045'  '\123'
  - escape sequences

\b \u0008  back space
\t \u0009  tab
\n \u000A  line feed
\f \u000C  form feed
\r \u000D  carriage return
\" \u0022
\' \u0027
\\ \u005c
Literals

- String literals
  - "" "\"" "this is a String"
- null literal